Network Servers and Java

J. Franco*
Computer Science Division, ECECS
University of Cincinnati
Cincinnati, OH 45221-0030
E-mail: franco@gauss.ececs.uc.edu

December 17, 1996

Even before the Web, network *servers* played a central role on the internet by facilitating E-mail, GOPHER, telnet, and FTP usage. Additional servers, including those supporting the *Hypertext Transfer Protocol* (*HTTP*), have been developed recently to handle Web traffic. This protocol and the development of the computer language called Java have raised our consciousness of network servers. Many of us know that servers are involved with the movement of information over the net but are not sure exactly what they are and how they function. We have seen servers installed on machines but, perhaps despite a nagging curiosity, have left them to the system administrators for fear of getting in over our heads. The purpose of this article is to explain the rudiments of network servers and show how they can be easily implemented in Java. In particular, we implement a simple HTTP server that can be put on a DOS or UNIX platform for experimentation.

1 Point to Point Transfer of Information

Servers and clients facilitate the flow of information between computers. A *client* such as Netscape can be used to initiate a query that is sent to an HTTP server operating on, say, the host computer at the university. The query might represent a request for a specific document containing admissions data. The server locates the document and sends its contents back to the client. In loading the document for viewing, the client notices additional files, for example images, need to be loaded. It makes further requests to the server and the server responds until all needed files are received by the client and loaded.

2 HTTP

Both client and server pass messages to each other in accordance with a language of communication that is designed to eliminate ambiguities in requests and to be independent of hardware. Such a

*Supported in part by the Office of Naval Research, N00014-94-1-0382

language is called a *protocol.* Most Web browsers support several protocols including HTTP, FTP, and GOPHER. Each has a specific purpose. Here we will illustrate an exchange in HTTP which is a simple but very useful application-level protocol.

HTTP is primarily used to transfer hypermedia files and Web documents written in the Hypertext Markup Language (HTML). A client can request a page, say named `index.html`, from an HTTP server with a plain-ascii GET method request header such as the following:

```
GET /index.html HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/3.0 (Win95; I)
Host: localhost:80
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
```

The server responds to the client with a plain-ascii header and document such as the following:

```
HTTP/1.0 200 OK
Date: Sun, 14 Dec 1996 10:18:59 GMT
Server: Apache/1.0.0
Content-type: text/html
Content-length: 1579
Last-modified: Mon, 22 Jul 1996 22:23:34 GMT

< ...contents of index.html... >
```

Headers can include additional lines such as `Authorization:` and `WWW-Authenticate:` which allow access controls to be placed on the documents, and `Referer:` which enables a server to trace a client through a series of requests. Some of the header lines shown above are not needed but the `Content-type:` and `Content-length:` lines of the response header and the first line of the GET request are mandatory. Two other methods of transfer are the POST and HEAD methods. For a detailed specification of HTTP including all methods and header possibilities the reader is referred, on the Web, to

http://www.freesoft.org/Connected/RFC/1945/index.html

3 TCP/IP

At application-level, HTTP is used to request and receive Web documents. However, layered below this protocol are other protocols that are responsible for making the connection between computers so they can communicate with each other.

The Internetwork Protocol (IP) provides the fundamental data transfer services for the internet. Unlike early telephone systems, a connection between two computers is not fixed during the time the connection is made or even during a single, long transmission of data. Data handling on the internet resembles the handling of mail by the U.S. Post Office. A parcel is given an address and the interstate highway system and secondary roads are used to move it to and between intermediate collection points and then, finally, to the location corresponding to the address. On the internet, data is assembled at the originating process as a *packet* or *datagram*, usually of some bounded size such as 2000 bytes, with a header containing a destination address. In the case of a long transmission, many packets may have to be assembled. After being sent out, a packet is routed, depending on congestion and other factors, along several pathways and eventually is delivered to the specified destination. The packet makes the journey through many intermediate computers which check the address and send the packet one hop closer to its destination. The Internetwork Protocol is used to make this flow of data happen.

The Transmission Control Protocol (TCP) is layered on top of IP and provides a *stream*-oriented data transmission abstraction for network applications. At application-level it is convenient to think of two communicating processes on different computers as directly connected to end points of a channel. Over this channel, a stream of bytes sent from one process to its connected counterpart should arrive completely intact and in order. In actuality, many packets may be sent from one process to the other process and all may take different routes and arrive out of order. Some packets may not arrive at all. It is the job of the TCP layer to reorder data and request retransmission of lost data so the stream abstraction can be utilized. This is accomplished with the help of buffers at each endpoint. We use the terms `port` and `socket` when referring to an endpoint. These terms mean almost the same thing: the difference is for each port, there may be many sockets. Thus, a server can "listen" for messages on a single port and have simultaneous stream connections to many machines through multiple sockets on that port. In this way the TCP/IP connection, hidden from the application-level coder, provides for the reliable and predictable transfer of data between sockets.

4 Uniform Resource Locators

At this time it is instructive to present a specific Web location and show which parts are handled by which protocols. A typical, complete Uniform Resource Locator (URL) supplied by a client is

```
http://www.ececs.uc.edu:80/department/programs/index.html
```

This URL specifies the application-level protocol (HTTP), the server host name (www.ece.uc.edu) the port the server is listening on (80), and the path within the server host specifying the location of a document on the server host. The client sees this is an HTTP communication and sends a properly

formatted GET request containing the path along with the server host and port information to TCP/IP. TCP/IP uses the host and port information to make a "connection" to the intended server. The server is then handed the GET request by TCP/IP. Defaults are possible. For example, port 80 will be the default destination if `:80` is left out of the line above (the default port for HTTP) and index.html will be the default file name if `index.html` is left out of the line.

5 An HTTP server in Java

Java is a general purpose programming language having approximately the same utility as the C++ language. It is based on the object-oriented paradigm. In this paradigm, classes of objects derived from other classes inherit the methods of those classes. Java supplies a number of classes related to streams and connections to ports and sockets. Therefore, Java can be used easily to implement network servers. In this section we implement in Java a simple HTTP server supporting only a part of the GET method.

A server can be attached to a port with the following line of code:

```
ServerSocket server = new ServerSocket(80);
```

where `server` is declared an object of class ServerSocket and is created with value 80. Thus, `server` is "listening" on port number 80. We chose the number 80 because that is the default port number for HTTP. If this server is run on a machine with an already running HTTP server on port 80, a Java exception will be raised. In that case, some other port number should be chosen. We want the server to "listen" on port 80 until a message is received from some calling process. Then it should wake up and create a socket for a connection to the requesting client. This is accomplished with the following line of code:

```
Socket client = server.accept();
```

The Socket object `client` will be used to set up the two-way stream connection. This is accomplished using the following lines of code:

```
PrintStream os = new PrintStream(client.getOutputStream());
DataInputStream is = new DataInputStream(client.getInputStream());
```

Now a GET request can be read from the input stream with the following line of code:

```
String istring = is.readLine();
```

This line receives bytes from the input stream `is` until the newline character is reached and passes all the bytes read to a String object called `istring`. Since we are implementing an HTTP server, the validity of the received string needs to be checked to make sure it is a GET request. At the same time, the string can be parsed for the location of the requested file. The following function represents one of many ways to do this:

```
static String getPath(String msg)
{
    if (msg.length() == 0 || msg.substring(0,3) == "GET") return null;
    String path = msg.substring(msg.indexOf(' ')+1);
    path = path.substring(0, path.indexOf(' '));
    if (path.equals("")) return "index.html";
    if (path.charAt(path.length()-1) == '/') path += "index.html";
    return path;
}
```

First the `GET` and `HTTP/1.0` are stripped from the request message `msg`. What remains is contained in `path`. If `path` has zero length, `index.html` is returned. Otherwise, `path` begins with `/`. If `path` ends with `/` then `index.html` is appended to it and `path` is returned. Otherwise, `path` is returned as is. All that remains to be done is to open the requested file and send the header and the file contents to the client. Transmission of the header is accomplished by the following function:

```
static void fileFoundHeader(PrintStream os, int filelength)
{
    os.print("HTTP:/1.0 200 OK\n");
    os.print("Content-type:  text/html\n");
    os.print("Content-length:  "+filelength+"\n");
    os.print("\n");
}
```

which dumps the header to the client's input stream. The length of the requested file is passed and used in the `Content-length:` line. We have put only necessary lines into the header. The following function dumps the contents of the requested file onto the output stream.

```
static void sendReply(PrintStream os, DataInputStream in, int flen)
{
    try
    {
        byte buffer[] = new byte[flen];
        in.readFully(buffer);
        os.write(buffer, 0, flen);
        in.close();
    }
    catch (Exception e) { System.out.println(e); }
}
```

In case the requested file is not found or is read-protected, the following function returns an error message:

```
static void errorHeader(PrintStream os, String errmessage)
{
    os.print("HTTP:/1.0 404 Not Found\n");
    os.print("Content-type:  text/html\n");
    os.print("Content-length:  "+errmessage.length()+"\n");
    os.print("\n");
    os.print(errmessage+"\n");
}
```

Because our server will only look at the first line of a GET request, it will be unable to determine when the leading / in that line is needed and when it is superfluous. Therefore, we need the following function that attempts to open a path with no leading / if it is found that the path with the leading / does not exist.

```
static File OpenFile(String filename)
{
    File file = new File(filename);
    if (file.exists()) return file;
    if (filename.charAt(0) != '/') return file;
    return new File(filename.substring(1));
}
```

All of the functions above may be added to a Java class called `WebServe` that makes use of them as follows:

```
import java.net.*;
import java.io.*;

public class WebServe
{
    // Put above functions here

    public static void main(String arg[])
    {
        String path; DataInputStream in, is; PrintStream os;
        try
        {
            ServerSocket server = new ServerSocket(80);
            while (true)
            {
                Socket client = server.accept();
                os = new PrintStream(client.getOutputStream());
                is = new DataInputStream(client.getInputStream());
                if ((path = getPath(is.readLine())) != null)
                {
                    File file = OpenFile(path);
                    if (file.exists())
                    {
                        try
                        {
                            in = new DataInputStream(new FileInputStream(file));
                            fileFoundHeader(os, (int)file.length());
                            sendReply(os, in, (int)file.length());
                        }
                        catch (Exception e) {
                            errorHeader(os, "<h2>Can't Read "+path+"</h2>");
                        }
                        os.flush();
                    }
                    else
                        errorHeader(os, "<h2>Not Found "+path+"</h2>");
                }
                client.close();
            }
        }
        catch (IOException e) { System.out.println(e); }
    }
}
```

This code can be used on a computer with a Java interpreter and internet access. **However, you should not try to run it if you are at all concerned about other processes reading your computer's files.** The code should be placed in a file named `WebServe.java` and compiled using the Java development kit as follows: `javac WebServe.java`. This produces the file `WebServe.class`. To run it use the command `java WebServe`. In another shell, in the same directory that the server was run from, create a file named `index.html` with the contents `<h1>Hello Web</h1>`. Test the server by opening Netscape or other Web browser. Open the URL `http://localhost:80/`. The result should be a rather large `Hello Web` in the viewing window.

The server presented here has many deficiencies that present opportunities for learning. The most serious is security - there isn't any. On a DOS machine most of your files will be readable by anyone. Assuming the server is running from directory `/Java/WebServer`, opening the URL `http://localhost:80/../../autoexec.bat` presents a shocking picture of what can happen. Another problem is the server can handle only one client at a time. Java has remedies for these defects but space does not permit a discussion of these here.

6 Read More About It

1. To learn about Java consult the Java Web pages at `http://java.sun.com/`.
2. To learn more about internet protocols and other aspects of the internet consult the internet encyclopedia at `http://www.freesoft.org/Connected/`.
3. To learn about and download actual HTTP servers consult `http://hoohoo.ncsa.uiuc.edu/` and `http://www.apache.org/`.
4. To learn about the common gateway interface (CGI) consult `http://www.w3.org/pub/WWW/CGI/Overview.html`.
5. For a list of servers world-wide consult `http://www.cyf-kr.edu.pl/www-serw/wwwog.html`.